

AD-A038 867

STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE  
COMPLEXITY RESULTS FOR BANDWIDTH MINIMIZATION.(U)  
JAN 77 M R GAREY, R L GRAHAM, D S JOHNSON

F/6 9/4

N00014-76-C-0330

UNCLASSIFIED

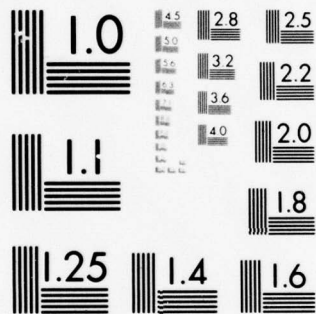
STAN-CS-77-587

NL

| OF |  
AD  
A038867



END  
DATE  
FILMED  
5-77



MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A

AD A 038867

12  
nw

COMPLEXITY RESULTS FOR BANDWIDTH MINIMIZATION

by

M. R. Garey, R. L. Graham, D. S. Johnson, & D. E. Knuth

STAN-CS-77-587  
JANUARY 1977



COMPUTER SCIENCE DEPARTMENT  
School of Humanities and Sciences  
STANFORD UNIVERSITY

AD No. \_\_\_\_\_  
DDC FILE COPY,

**DISTRIBUTION STATEMENT A**  
Approved for public release;  
Distribution Unlimited



See 1473

Unclassified

9 Technical rept.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 14 STAN-CS-77-587	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 COMPLEXITY RESULTS FOR BANDWIDTH MINIMIZATION.	5. TYPE OF REPORT & PERIOD COVERED technical, January 1977	
7. AUTHOR(s) 10 Michael Ronald David R. Garey, R. L. Graham, A. S. Johnson & D. E. Knuth	6. PERFORMING ORG. REPORT NUMBER STAN-CS-77-587	
9. PERFORMING ORGANIZATION NAME AND ADDRESS Stanford University Computer Science Department Stanford, Ca. 94305	8. CONTRACT OR GRANT NUMBER(s) 15 NO0014-76-C-0330 NSF-MCS-72-03752	
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Department of the Navy Arlington, Va. 22217	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) ONR Representative: Philip Surra Durand Aeronautics Bldg., Rm. 165 Stanford University Stanford, Ca. 94305	12. REPORT DATE 11 January 1977	
	13. NUMBER OF PAGES 36 12380.	
	15. SECURITY CLASS (of this report) Unclassified	
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE		
16. DISTRIBUTION STATEMENT (of this Report) Releasable without limitations on dissemination		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) bandwidth, directed bandwidth, linear algorithm, NP-complete problems, optimum permutations, siphonophora		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)  We present a linear-time algorithm for sparse symmetric matrices which converts a matrix into pentadiagonal form ("bandwidth 2"), whenever it is possible to do so using simultaneous row and column permutations. On the other hand when an arbitrary integer k and		

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

graph  $G$  are given, we show that it is NP-complete to determine whether or not there exists an ordering of the vertices with bandwidth  $\leq k$ , even when  $G$  is restricted to the class of free tress with all vertices of degree  $\leq 3$ . Related problems for acyclic directed graphs (upper triangular matrices) are also discussed.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

# Complexity Results for Bandwidth Minimization

by Michael R. Garey,<sup>\*/</sup> Ronald L. Graham,<sup>\*/</sup> David S. Johnson,<sup>\*/</sup>  
and Donald E. Knuth

Computer Science Department  
Stanford University  
Stanford, California 94305

## Abstract.

*is described.* We present a linear-time algorithm for sparse symmetric matrices which converts a matrix into pentadiagonal form (bandwidth  $2n$ ), whenever it is possible to do so using simultaneous row and column permutations. On the other hand when an arbitrary integer  $k$  and graph  $G$  are given, we show that it is NP-complete to determine whether or not there exists an ordering of the vertices with bandwidth  $\leq k$ , even when  $G$  is restricted to the class of free trees with all vertices of degree  $\leq 3$ . Related problems for acyclic directed graphs (upper triangular matrices) are also discussed.

Keywords: bandwidth, directed bandwidth, linear algorithm, NP-complete problems, optimum permutations, siphonophora.

<sup>\*/</sup> Bell Telephone Laboratories, Murray Hill, New Jersey 07974.

This research, performed in part while the first three authors were visiting Stanford University, was supported by National Science Foundation grant MCS 72-03752 A03 and by the Office of Naval Research contract N00014-76-C-0330. Reproduction in whole or in part is permitted for any purpose of the United States Government.

ACCESSION for	
NTIS	White Section
DOC	Ref Section
UNANNOUNCED	Per
JUSTIFICATION	7-14-73
BY	Atch.
DISTRIBUTION/AVAILABILITY CAC	
USE	AVAIL. and/or SPEC
N	

# 1. Introduction.

Let  $G$  be a graph on the set of vertices  $V$ , where  $|V| = n$ . We shall write  $u - v$  if vertex  $u$  is adjacent to vertex  $v$  in  $G$ , and  $u \not- v$  if they are not adjacent. A layout of  $G$  is a one-to-one mapping  $f$  that takes  $V$  into the positive integers; equivalently, a layout can be regarded as a string of vertices and "blanks", with each vertex of  $V$  appearing exactly once, for instance  $b\_c\_da$ . The correspondence between these two definitions is simply that  $f(v) = k$  if and only if  $v$  is the  $k$ -th element of the string; thus  $b\_c\_da$  corresponds to  $f(a) = 7$ ,  $f(b) = 1$ ,  $f(c) = 3$ ,  $f(d) = 6$ , where  $V = \{a, b, c, d\}$ .

The bandwidth of a layout  $f$  is defined to be

$$\text{bandwidth}(f) = \max\{|f(u) - f(v)| : u - v\},$$

the greatest distance between  $G$ -adjacent vertices in the string corresponding to  $f$ . The bandwidth of graph  $G$  is then

$$\text{Bandwidth}(G) = \min\{\text{bandwidth}(f) : f \text{ is a layout of } G\}.$$

It is clear that

$$\text{Bandwidth}(G) = \max\{\text{Bandwidth}(G') : G' \text{ is a connected component of } G\},$$

for if  $f$  is any layout there is another layout  $f'$ , having the same bandwidth, in which the connected components of  $G$  appear "unmixed" as substrings. (We can let  $f'(v) = f(v) + Nc(v)$ , for example, where  $c(v)$  is the number of the component containing  $v$ , and where  $N$  is sufficiently large.)

Perhaps the most important application of the bandwidth notion arises in connection with sparse matrices. Given a sparse  $n \times n$  matrix  $A = (a_{ij})$ , let  $G$  be the graph on vertices  $\{v_1, \dots, v_n\}$  where  $v_i - v_j$  for  $i \neq j$  if and only if  $a_{ij} \neq 0$  or  $a_{ji} \neq 0$ . Then  $\text{Bandwidth}(G) \leq k$  if and only if there is a permutation matrix  $P$  such that all elements of  $P^T A P$  lie on the diagonal or on one of the first  $k$  superdiagonals or the first  $k$  subdiagonals. This is easily proved by observing that blanks may be removed from a layout without increasing the bandwidth.

When  $G$  has no edges, its bandwidth is trivially  $-\infty$ . Otherwise the bandwidth will be as low as 1 if and only if each component of  $G$  is an isolated point or a path, namely a subgraph of the form  $v_1 - v_2 - \dots - v_n$ , where  $v_i - v_j$  iff  $|i-j| = 1$ . It is easy to determine whether or not  $\text{Bandwidth}(G) = 1$ , even when  $G$  is not known to be connected, in linear time; in other words, there is an algorithm which decides in  $O(n)$  steps whether or not a sparse matrix can be converted into tridiagonal form by simultaneous row and column permutations. (See [13].) The simplicity of this algorithm suggests naturally that the next harder case might not be too difficult, and indeed we shall see below that the condition  $\text{Bandwidth}(G) = 2$  can be tested in linear time. However, the algorithm which achieves this is quite intricate, and there appears to be no elegant way to characterize graphs of bandwidth 2.

The authors have been unable to construct a polynomial-time algorithm that decides whether or not  $\text{Bandwidth}(G) = 3$ . The bandwidth 2 case indicates some of the difficulties which must be surmounted. Section 8 below shows that the general problem of deciding whether or not  $\text{Bandwidth}(G) \leq k$ , given  $k$ , is NP-complete, even if  $G$  is a free tree with all vertices of degree  $\leq 3$ . This restriction to trees is of special interest because the analogous problem of minimizing  $\sum |f(u) - f(v)|$  instead of  $\max |f(u) - f(v)|$  over all layouts can be done in polynomial time when the graph is a free tree [31], yet it is NP-complete for general graphs [17].

Section 9 considers the analogous problems which arise when acyclic directed graphs replace undirected graphs. Several open problems conclude the paper.

## 2. Preliminaries for the Algorithm.

In this section we shall begin to develop an algorithm that tests whether or not  $\text{Bandwidth}(G) = 2$ . We shall assume that  $G$  is connected and that it has at least one vertex of degree  $\geq 3$ . (If all vertices are of degree  $\leq 2$ , it is easy to see that  $\text{Bandwidth}(G) \leq 2$ , since such a graph is a collection of isolated points, paths, and cycles.) The connectedness assumption implies that  $G$  has at least  $n-1$  edges, and on the other hand we may assume that  $G$  has at most  $2n-3$  edges since a graph of bandwidth  $k$  cannot have more than  $(n-1) + (n-2) + \dots + (n-k)$  pairs of adjacent vertices. Therefore our algorithm will take  $O(n)$  steps if its running time is bounded by a constant times the number of edges in  $G$ .

In order to get into the right frame of mind for this problem, the reader is urged to try his or her hand at finding a bandwidth-2 layout for the graph in Figure 1. Like all graphs of bandwidth 2, this one is rather "skinny"; a breadth-first search will not involve many unexplored nodes at any time. The puzzle which the reader is now asked to try is simply this: Arrange the 27 vertices of Figure 1 into a straight line so that all pairs of vertices which are directly linked in that graph are separated by at most one other vertex in the line. (This puzzle is not quite so easy as it looks. The algorithm we shall develop is supposed to work in linear time, essentially without backing up, but no such restriction is being imposed on the reader.)

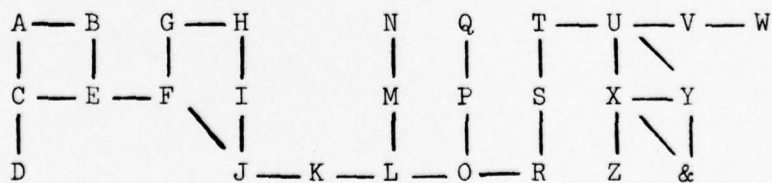


Figure 1. Example of a graph which the reader is urged to arrange into a bandwidth-2 layout before proceeding further.

Perhaps the most important notion which arises in connection with graphs of bandwidth 2 is the concept of chains within  $G$ . We say that  $v$  begins a chain of length  $k$  if there are vertices  $v = v_1, \dots, v_k$  such that

$$v_1 - v_2 - \dots - v_k$$

in  $G$ , and each of  $v_1, \dots, v_{k-1}$  has degree 2; furthermore  $v_k$  must be of degree 1, an endpoint.

Let us define  $l(v) = 1$  if  $\deg(v) = 1$ , and  $l(v) = k+1$  if  $\deg(v) = 2$  and  $v - w$  where  $l(w) = k$ ; otherwise  $l(v) = \infty$ . This function is well-defined since  $\text{Bandwidth}(G) > 1$ ; and it is clearly possible to compute  $l(v)$ , for all  $v$ , in  $O(n)$  steps. Therefore our algorithm will assume that this precomputation has been carried out. The values of  $l$  for the example graph in Figure 1 are shown in Figure 2. Note that vertex  $v$  is part of a chain if and only if  $l(v) < \infty$ .



Figure 2. The  $l$  function for the example graph in Figure 1; there are three chains of length 2.

We shall say that a layout  $f$  is chain-stretched if  $|f(v_i) - f(v_{i+1})| = 2$  whenever  $v_i$  and  $v_{i+1}$  are consecutive vertices of a chain. This terminology is justified because of the following observation.

Lemma. Every graph of bandwidth  $\leq 2$  has a chain-stretched layout of bandwidth  $\leq 2$ .

Proof. Let  $f$  be a layout for the graph  $G$ , where  $\text{Bandwidth}(G) \leq 2$ ; we may assume that  $G$  is connected. Furthermore we shall choose  $f$  to have the maximum "range span" over all bandwidth-2 layouts for  $G$ ; i.e.,

$\max_{v \in V} f(v) - \min_{v \in V} f(v)$  is to be maximum over all  $f$  with  $\text{bandwidth}(f) \leq 2$ . (The maximum range span is finite, at most  $2n-2$ , since  $G$  is connected.) We shall prove that  $f$  is chain-stretched.

If not, the string  $\phi$  corresponding to  $f$  contains the substring  $uv$ , where  $u$  and  $v$  are consecutive vertices of a chain. By definition,  $\deg(u)$  and  $\deg(v)$  are at most 2, and  $u - v$ , hence  $u$  and  $v$  are each adjacent to at most one other vertex. By maximality of  $f$ 's range span, the strings obtained from  $\phi$  by replacing  $uv$  by  $u_v$  and  $v_u$  are not layouts of bandwidth  $\leq 2$ . It follows that  $\phi$  contains the substring  $uvab$  or  $abuv$ , where  $a - u - v - b$ ; by left-right symmetry we may assume that  $\phi$  contains  $abuv$ . Then  $v$  must be the rightmost nonblank element of  $\phi$ . If  $\ell(u) > \ell(v) = k$ , graph  $G$  contains the chain  $v_k - v_{k-1} - \dots - v_1$  where  $v = v_k$  and  $b = v_{k-1}$ ; but then  $\phi$  must end with  $v_1 u_2 \dots u_{k-1} v_{k-1} u_k v_k$  and it can be lengthened by replacing this substring by  $u_2 \dots u_{k-1} u_k v_k v_{k-1} \dots v_1$ . On the other hand if  $\ell(v) > \ell(u) = k$ , a similar argument shows that  $\phi$  ends with  $u_1 v_1 u_2 \dots u_{k-1} v_{k-1} u_k v_k$  where  $u = u_k$  and  $a = u_{k-1}$ , and this substring can be replaced by  $v_1 \dots v_k u_k u_{k-1} \dots u_1$ . In both cases the maximality of range span has been contradicted.  $\square$

The algorithm we shall develop below is based on a subalgorithm which solves the following problem: "Given a connected graph  $G$  and two vertices  $a$  and  $b$ , decide whether or not there exists a layout  $f$  of bandwidth  $\leq 2$  such that  $f(a) = 1$  and  $f(b) = 2$ ." If such a layout beginning with  $ab$  exists, the algorithm will construct one; and in all cases the algorithm will terminate after  $O(n)$  steps. The idea is to build the layout step by step, working with partial layouts, namely with one-to-one functions  $f$  that are defined only on a subset of the vertices. All partial layouts we shall deal with will satisfy the bandwidth 2 condition, in the sense that  $|f(u) - f(v)| \leq 2$  whenever  $f(u), f(v)$  are both defined and  $u - v$ . Furthermore we know by the lemma that it suffices to restrict attention to chain-stretched partial layouts.

If  $f$  is a partial layout defined on the set of vertices  $U$ , the active vertices of  $f$  are those elements  $u \in U$  such that  $u - v$  for some  $v \notin U$ . If  $f_1$  is a partial layout defined on  $V_1$  and  $f_2$  is a partial layout defined on  $V_2 \supseteq V_1$ , we say that  $f_2$  is an extension of  $f_1$  if  $f_2(v) = f_1(v)$  for all  $v \in V_1$ . We also say that  $f_2$  is a complete layout if  $V_2 = V$  and  $\text{bandwidth}(f_2) \leq 2$ . Thus the task of our subalgorithm will be to decide whether or not the partial layout  $f$  defined by the string  $ab$  (i.e.,  $f(a) = 1$  and  $f(b) = 2$ ) can be extended to a complete layout.

The subalgorithm actually does more, since its initial task leads to a family of similar subtasks of three types:

- Type A. Given a partial layout defined by the string  $\alpha ab$ , where at most  $a$  and  $b$  are active, can it be extended to a complete layout?
- Type B. Given two partial layouts defined by the strings  $\alpha a_m b_m \dots a_1 b_1$  and  $\alpha b_m a_m \dots b_1 a_1$ , for some  $m \geq 1$ , where at most  $a_1$  and  $b_1$  are active, can at least one of these be extended to a complete layout?
- Type C. Given a partial layout defined by the string  $\varphi = \alpha a_m \dots a_1$  for some  $m \geq 1$ , where at most  $a_1$  is active, can it be extended to a complete layout?

In each case  $\alpha$  is a (possibly empty) initial string which has no important influence on the algorithm, since it represents inactive vertices and blanks that have already been permanently placed. The string  $\alpha$  in tasks of Type C will have length  $\geq 2$ , and its final two elements will be nonblank. The two strings in tasks of Type B will be denoted by  $\varphi = \alpha \langle a_m b_m \rangle \dots \langle a_1 b_1 \rangle$ .

The idea of the subalgorithm is quite simple, namely to "keep doing something useful." Let  $f$  be a partial layout of one of the three types, defined on the vertices  $U$ . (Actually  $f$  represents two partial layouts if it is of Type B, but it will be convenient to ignore this fine distinction in our informal discussion.) By looking at how the active vertices of  $f$  interact with vertices  $\notin U$ , it may be obvious that  $f$  cannot be completed. Otherwise the subalgorithm will find a sufficiently general extension of  $f$ , namely an extension layout  $f'$

which can be completed whenever  $f$  can be; and  $f'$  will have one of the three basic types. If any suitable extension is found, the string  $\phi$  corresponding to  $f$  will be replaced by the string  $\phi'$  corresponding to  $f'$ , and the process will continue until either reaching an impasse or a complete layout. The running time for each extension step will be bounded, except in one case where the running time can be "charged" to subsequent extension steps; hence the total time will be  $O(n)$ .

In Section 7 we shall show how the subalgorithm can be used to construct an algorithm that solves the general bandwidth 2 problem (without any given partial layout), in linear time.

### 3. The Subalgorithm for Types A and B.

We shall present the subalgorithm informally, with proofs of the validity of each extension intermixed with specifications of the actual operations to be carried out. The actions will be of three kinds:

(a) Terminate successfully because  $\phi$  is complete; (b) terminate unsuccessfully because  $\phi$  cannot be completed; (c) set  $\phi'$  to a sufficiently general extension of  $\phi$ . It is hoped that this manner of presenting the procedure will make it easy to understand and reasonably enjoyable to read. Examples of the subalgorithm in operation appear in Section 6 below.

The following notation will be used for convenience:

$U$  = set of vertices appearing in  $\phi$  = domain of current partial layout  $f$  ;

$S(u) = \{v \mid u - v \text{ and } v \notin U\}$  = "successors" of vertex  $u$  ;

$n(u) = \|S(u)\|$  = number of "successors" of  $u$  ;

$l(u)$  = chain level of  $u$  (defined earlier).

It is clearly possible to build and maintain data structures so that references to  $S(u)$ ,  $n(u)$ ,  $l(u)$  take a bounded amount of time. The subalgorithm consists of a long but exhaustive list of cases covering which actions are appropriate under various circumstances that can arise.

First let us consider Type A, recalling that tasks of this type are specified by the string  $\phi = \alpha ab$ , where at most  $a$  and  $b$  are active.

Case A1,  $n(a) > 1$  or  $n(b) > 2$ . Failure.

Case A2,  $n(a) = 1$ . Set  $\phi' = \alpha abc$  where  $S(a) = \{c\}$ .

Case A3,  $n(a) = 0$ ,  $n(b) = 2$ . Set  $\phi' = \alpha ab\langle cd \rangle$  where  $S(b) = \{c, d\}$ .

Case A4,  $n(a) = 0$ ,  $n(b) = 1$ . Set  $\phi' = \alpha ab\_c$  where  $S(b) = \{c\}$ .

Case A5,  $n(a) = 0$ ,  $n(b) = 0$ . Success.

Note that Cases A2, A3, A4 lead to new problems of Type A, B, C respectively; the proofs of validity in each case are trivial.

Recall that tasks of Type B are specified by the string  $\varphi = \alpha \langle a_m b_m \rangle \dots \langle a_1 b_1 \rangle$ , for some  $m \geq 1$ , where at most  $a_1$  and  $b_1$  are active. Actually  $\varphi$  represents a potential choice between two partial layouts,  $\alpha a_m b_m \dots a_1 b_1$  and  $\alpha b_m a_m \dots b_1 a_1$ . For convenience we shall write  $a = a_1$ ,  $b = b_1$ ; we may assume by symmetry that  $n(a) \leq n(b)$ .

Case B1.  $\|S(a) \cup S(b)\| > 2$  or  $n(a) = n(b) = 2$ . Failure.

Case B2.  $n(a) = 1$ ,  $n(b) = 2$ . Set  $\varphi' = \alpha a_m b_m \dots a_1 b_1 c d$  where  $S(a) = \{c\}$  and  $S(b) = \{c, d\}$ .

Case B3.  $n(a) = 0$ ,  $n(b) = 2$ . Set  $\varphi' = \alpha a_m b_m \dots a_1 b_1 \langle cd \rangle$  where  $S(b) = \{c, d\}$ .

Case B4.  $n(a) = 1$ ,  $n(b) = 1$ ,  $S(a) = S(b)$ . Set  $\varphi' = \alpha a_m b_m \dots a_1 b_1 c$  where  $S(a) = \{c\}$ .

Case B5.  $n(a) = 1$ ,  $n(b) = 1$ ,  $S(a) \neq S(b)$ . Set  $\varphi' = \alpha \langle a_m b_m \rangle \dots \langle a_1 b_1 \rangle \langle cd \rangle$  where  $S(a) = \{c\}$ ,  $S(b) = \{d\}$ .

Case B6.  $n(a) = 0$ ,  $n(b) = 1$ . Set  $\varphi' = \alpha a_m b_m \dots a_1 b_1 c$  where  $S(a) = \{c\}$ .

Case B7.  $n(a) = 0$ ,  $n(b) = 0$ . Success.

Again the proofs in each case are trivial; we shall discuss only case B6 here: Any completion of  $\varphi$  must be of the forms  $\alpha a_m b_m \dots a_1 b_1 x c w$  (where  $x$  is a vertex or a blank),  $\alpha a_m b_m \dots a_1 b_1 c w$ , or  $\alpha b_m a_m \dots b_1 a_1 c w$ . The first of these is an extension of  $\varphi'$ ; and the second or third imply that  $\alpha a_m b_m \dots a_1 b_1 c w$  is also a complete extension.

#### 4. The Subalgorithm for Type C.

Recall that tasks of Type C are specified by the string  $\varphi = \alpha \_a_m \dots \_a_1$ , for some  $m \geq 1$ , where at most  $a_1$  is active and  $\alpha$  contains no usable blanks. This type of partial layout allows considerably more flexibility than Types A and B do, since it may be possible to make good use of the  $m$  blanks. Let us write  $a$  as a shorthand for  $a_1$ . Furthermore we shall write  $U' = U \cup S(a)$ , with  $S'(u)$  and  $n'(u)$  defined correspondingly.

Case C1,  $n(a) > 3$ . Failure.

Case C2,  $S(a) = \{b, c, d\}$ .

In this case the final neighborhood of  $a$  in a complete extension must be  $bacd$ ,  $badc$ ,  $cabd$ ,  $cadb$ ,  $dabc$ , or  $dacb$ ; the possibilities can be narrowed down by considering various subcases. Symmetry between  $b, c, d$  is used in order to reduce the number of possibilities; in other words, there is always a way to rename the elements of  $S(a)$  so that some subcase applies. We shall say that a vertex  $u$  in  $S(a)$  is feasible if it can conceivably fit to the left of  $a_1$ ; thus  $u$  is feasible if  $S'(u) = \{v\}$  where  $\ell(v) < m$ , or if  $n'(u) = 0$ . In the former case we say that  $u$  is  $\ell(v)$ -feasible; in the latter case we say that  $u$  is 0-feasible.

Case C2.1,  $b - c, b - d, c - d$ . Failure.

Case C2.2,  $b \neq c, b - d, c - d$ .

In this case we must decide between  $badc$  and  $cadb$ .

Case C2.2.1, neither  $b$  nor  $c$  is feasible. Failure.

Case C2.2.2,  $b$  is feasible but not  $c$ . Set  $\varphi' = \alpha[ba]dc$ .

Here and in the sequel we shall use the following notation:

$[ba] = \_a_m \dots \_a_{k+2} b_k a_{k+1} \dots b_0 a_1$  if  $b = b_0$  is  $k$ -feasible and  $b_1 - \dots - b_k$  is the corresponding chain of length  $k$ . In other words,  $[ba]$  stands for the string  $\_a_m \dots \_a_1$  with  $b$  and its successors inserted into the appropriate blank spaces.

Case C2.2.3,  $b$  is  $k$ -feasible and  $c$  is  $\ell$ -feasible where  $k \geq \ell$ . Set  $\varphi' = \alpha[ba]dc$ .

To justify this step, we shall prove that

$$\alpha[ba]dc \geq \alpha[ca]db ,$$

where we say that partial layout  $\varphi_1$  dominates  $\varphi_2$  (written  $\varphi_1 \geq \varphi_2$ ) if every completion of  $\varphi_2$  implies the existence of a completion of  $\varphi_1$ . In our case any chain-stretched completion of  $\varphi$  which is not an extension of  $\varphi'$  must be an extension of  $\alpha[ca]db$ , so it must have the form  $\varphi'' = \alpha[ca]d_0b_0d_1b_1 \dots d_kb_k\omega$ . Let  $c_0 - c_1 - \dots - c_l$  be the chain adjacent to  $c = c_0$  and let  $c_j$  be blank if  $l < j \leq k$ . Then we may interchange  $c_0, \dots, c_k$  with  $b_0, \dots, b_k$  in  $\varphi''$ , obtaining a valid completion of  $\varphi$  which extends  $\varphi'$ .

It is important that the reader understand the justification of step C2.2.3 at this point before proceeding further. Although the argument is very simple, we shall be using it repeatedly in the sequel, with various refinements and extensions as the cases get more complex.

Case C2.3,  $b - c$ ,  $b + d$ ,  $c + d$ .

In this case we must decide between  $bacd$ ,  $cabd$ ,  $dabc$ , and  $dacb$ .

Case C2.3.1, neither  $b$  nor  $c$  is feasible. Failure, unless  $d$  is feasible. In the latter case, set  $\varphi' = \alpha[da]\langle bc \rangle$ .

Case C2.3.2,  $b$  is feasible but not  $c$ ; say  $b$  is  $k$ -feasible. If  $d$  is  $l$ -feasible where  $l \geq k$ , set  $\varphi' = \alpha[da]bc$ , otherwise set  $\varphi' = \alpha[ba]cd$ .

To justify this step, note that  $\alpha[ba]cd$  is forced unless  $d$  is feasible. In the latter case  $\alpha[da]cb$  cannot be better than  $\alpha[da]bc$ , since  $b = b_0$  must be followed by  $b_1, \dots, b_k$ , with  $b_{i+1}$  following two positions after  $b_i$ ; it is easy to see that any completion of  $\alpha[da]cb$  can be converted into one which extends  $\alpha[da]bc$ . Thus we must simply distinguish between  $bacd$  and  $dabc$ , and the argument is similar to Case C2.2.3.

Case C2.3.3,  $b$  is  $k$ -feasible and  $c$  is  $l$ -feasible, where  $k \geq l$ . Set  $\varphi' = \alpha[ba]cd$ .

The argument is like Case 2.2.3 again; if  $d$  is feasible too, we will soon be successful, regardless of which alternative is chosen.

Case C2.4,  $b \vdash c$ ,  $b \vdash d$ ,  $c \vdash d$ .

All six possibilities of Case C2 still remain, but we can make use of the symmetry.

Case C2.4.1, none of  $b$ ,  $c$ ,  $d$  is feasible. Failure.

Case C2.4.2,  $b$  is feasible but  $c$  and  $d$  are not. Set  $\varphi' = \alpha[ba]\langle cd \rangle$ .

Case C2.4.3,  $b$  is  $k$ -feasible and  $c$  is  $l$ -feasible, where  $l \leq k$ , but  $d$  is infeasible. Set  $\varphi' = \alpha[ba]cd$ .

In this case  $\alpha[ba]cd \geq \alpha[ba]dc$  and  $\alpha[ca]bd \geq \alpha[ca]db$  as in Case C2.3.2, while  $\alpha[ba]cd \geq \alpha[ca]bd$  as in Case 2.2.3.

Case C2.4.4, all of  $b$ ,  $c$ ,  $d$  are feasible. Set  $\varphi' = \alpha[ba]cd$ . Success is imminent.

Case C3,  $S(a) = \{b, c\}$ . See Section 5.

This is by far the hardest case to handle, and we shall postpone it for a moment since the remaining cases are very simple.

Case C4,  $S(a) = \{b\}$ . Set  $\varphi' = \alpha_{a_n} \dots a_1 b$ .

This clearly dominates  $\alpha_{a_n} \dots a_2 b a_1$  and  $\alpha_{a_n} \dots a_1 b$ .

Case C5,  $n(a) = 0$ . Success.

5. The Subalgorithm for Type C, Case C3.

Now we must face up to Case C3; as above we have  $\varphi = \alpha_{a_m} \dots a_1$  and  $a = a_1$  and  $S(a) = \{b, c\}$ . We should replace the substring  $a$  at the right of  $\varphi$  by either  $abc$ ,  $acb$ ,  $bac$ ,  $ba_c$ ,  $cab$ , or  $ca_b$ , where the dashes may or may not get filled in later. Fortunately we can rule out two of these possibilities immediately, since  $bac$  is never better than  $abc$  and  $cab$  is (similarly) never better than  $acb$ : The complete layout  $\alpha[ba]c\omega$  which extends  $bac$  can always be converted to a complete layout  $\alpha[b_1a]bc\omega$  which extends  $abc$ .

Case C3.1,  $b - c$ .

In this case we have to distinguish between  $abc$  and  $acb$ . Let us say that  $b$  is k-lucky if  $S'(b)$  contains a vertex  $b_1$  with  $\ell(b_1) = k$  and  $k \leq m$ . (If there are two or more such vertices  $b_1$ , choose one with maximum  $k$ .) Similarly  $c$  might be lucky; we can use the blanks left of  $a$  for one of the successors of a lucky vertex.

Case C3.1.1, neither  $b$  nor  $c$  is lucky. Set  $\varphi' = \alpha_{a_m} \dots a_1 \langle bc \rangle$ .

Case C3.1.2,  $b$  is  $k$ -lucky and  $c$  is either (i) unlucky or (ii)  $\ell$ -lucky where  $\ell < k$ , or (iii)  $k$ -lucky and  $n'(b) \leq n'(c)$ . Set  $\varphi' = \alpha[b_1a]bc$ .

To justify this step, we first argue (as in Case C2.2.3) that the layout  $\alpha_{a_m} \dots a_1 bcb_1$  has no advantage over  $\varphi'$ . Therefore the only competing possibility is  $\alpha_{a_m} \dots a_1 cb$ . By considering the two ways to place  $b_1$  in the latter string, we have two possible types of completion to consider, say  $\varphi'' = \alpha[c_1a]cbx_1b_1 \dots x_kb_k\omega$  and  $\varphi''' = \alpha[c_1a]cbb_1x_1 \dots x_{k-1}b_{k-1}\omega$ , since  $b_1$  has degree  $\leq 2$  and is part of a stretched chain. (Here  $c_1$  is blank if  $c$  is unlucky or if we do not choose to make use of  $c$ 's luckiness.) We can always replace  $\varphi''$  by  $\alpha[b_1a]bcx_1c_1 \dots x_kc_k\omega$ , an extension of  $\varphi'$ ; similarly,  $\varphi'''$  can always be replaced by  $\alpha[b_1a]bcx_1c_1 \dots x_{k-1}c_{k-1}\omega$  unless  $c$  is  $k$ -lucky. But in the latter case we have  $n'(b) \leq n'(c) = 1$  by hypothesis, so the  $x_i$  are all blank and  $\omega$  is empty;  $\varphi'''$  can therefore be replaced by  $\alpha[b_1a]bc_{c_1} \dots c_k$ .

Case C3.2,  $b \vdash c$  and  $n'(b) > 3$ . Failure.

Case C3.3,  $b \vdash c$  and  $n'(b) = 3$ . If  $S'(b) \cap S'(c) = \{d\}$  and either  $S'(c) = \{d\}$  or  $S'(c) = \{c_1, d\}$  where  $l(c_1) < m$ , set  $\varphi' = \alpha[ca]db$ . If  $S'(b) \cap S'(c) = \emptyset$  and either  $S'(c) = \emptyset$  or  $S'(c) = \{c_1\}$  where  $l(c_1) < m$ , set  $\varphi' = \alpha[ca]_b$ . Otherwise failure.

Case C3.4,  $b \vdash c$  and  $\max(n'(b), n'(c)) = 2$ .

Case C3.4.1,  $S'(b) = S'(c)$ . Failure.

Case C3.4.2,  $S'(b) \cap S'(c) = \{d\}$ . If  $n'(b) = 2$ , let  $S'(b) = \{b_1, d\}$ ; if  $n'(c) = 2$  let  $S'(c) = \{c_1, d\}$ .

In this case we say that  $b$  is  $k$ -lucky if  $l(b_1) = k$  and  $k \leq m$ ;  $b$  is 0-lucky if  $n'(b) = 1$ ; otherwise  $b$  is unlucky. Similarly  $c$  can be lucky or unlucky. There are four viable alternatives to decide between, namely  $\alpha[ba]dc$ ,  $\alpha[b_1a]bcd$ ,  $\alpha[ca]db$ , and  $\alpha[c_1a]cbd$ .

Case C3.4.2.1, neither  $b$  nor  $c$  is lucky. Failure.

Case C3.4.2.2,  $b$  is  $k$ -lucky and  $c$  is unlucky. If  $k = m$ , set  $\varphi' = \alpha[b_1a]bcd$ . Otherwise set  $\varphi' = \alpha_{-a_m} \dots \alpha_{-a_{k+2}} \langle b_k a_{k+1} \rangle \dots \langle ba_1 \rangle \langle dc \rangle$ .

This is the neatest part of the entire algorithm, since the two viable alternatives  $\alpha[ba]dc$  and  $\alpha[b_1a]bcd$  turn out to be essentially a Type B situation. (On the other hand it may also be considered the sloppiest part of the algorithm, since an abuse of notation is involved here: If the Type B specification is ultimately completed to a string of the form  $\alpha_{-a_m} \dots \alpha_{-a_{k+2}} a_{k+1} b_k \dots a_1 bcdw$ , a blank should actually be inserted just before  $a_{k+1}$ .)

Case C3.4.2.3,  $b$  is  $k$ -lucky and  $c$  is  $l$ -lucky, where  $k \geq l$ . Set  $\varphi' = \alpha[b_1a]bcd$ .

It is easy to check that  $\varphi'$  dominates the other three alternatives, using arguments like those in Section 4.

Case C3.4.3.  $S'(b) \cap S'(c) = \emptyset$  and  $n'(b) = n'(c) = 2$ . Let  $S'(b) = \{b_1, b'_1\}$  and  $S'(c) = \{c_1, c'_1\}$ , where  $l(b_1) \leq l(b'_1)$  and  $l(c_1) \leq l(c'_1)$ .

The only possibilities are  $\alpha[ba]b'_1cc_1c'_1$  and  $\alpha[ca]c'_1bb_1b'_1$ , perhaps interchanging  $b_1$  with  $b'_1$  and/or  $c_1$  with  $c'_1$ .

Case C3.4.3.1,  $l(b_1) \geq m$ . If  $l(c_1) \geq m$ , failure; otherwise if  $l(c'_1) \geq m$ , set  $\phi' = \alpha[ca]c'_1b$ ; otherwise set  $\phi' = \alpha[c'a]c_1b$ , where "[c'a]" means that the blanks are to be filled by  $c$  and the chain containing  $c'_1$ .

These actions are forced unless  $l(c_1) = l(c'_1) = 1$ , for if  $c_1$  and  $c'_1$  both have finite level we must have  $l(c_1) = 1$  or failure will be imminent.

Case C3.4.3.2,  $l(b_1) < m \leq l(b'_1)$ ,  $l(c_1) < m \leq l(c'_1)$ , and  $n'(b'_1) \leq n'(c'_1)$ . If  $S'(b'_1) \neq \{c'_1\}$ , failure; otherwise set  $\phi' = \alpha[ba]b'_1c$ .

In this case it is impossible to complete  $\phi$  with  $\alpha[ba]b'_1cc_1c'_1$ , since  $l(b'_1) > 1$ ; the only viable alternatives are  $\alpha[ba]b'_1cc'_1c_1$  and  $\alpha[ca]c'_1bb'_1b_1$ , and we must have  $b'_1 = c'_1$ . Now if  $S'(c'_1) \neq \{b'_1\}$ , the stated value of  $\phi'$  is forced, otherwise success is imminent.

Case C3.4.3.3,  $l(b_1) < m \leq l(b'_1)$  and  $l(c'_1) < m$ . Set  $\phi' = \alpha[c'a]c_1b$ .

This is essentially forced, since  $\alpha[ba]b'_1c\langle c_1c'_1 \rangle$  implies  $l(b'_1) = 1$  when  $c_1$  and  $c'_1$  have finite level.

Case C3.4.3.4,  $l(b'_1) < m$ ,  $l(c'_1) < m$ , and  $l(b_1) \leq l(c_1)$ . Set  $\phi' = \alpha[b'a]b_1c$ .

As in Case C3.4.3.1 we see that failure will occur unless  $l(b_1) = 1$ .

Case C3.4.4.  $S'(b) \cap S'(c) = \emptyset$ ,  $n'(b) = 2$ , and  $n'(c) \leq 1$ . Let  $S'(b) = \{b_1, b'_1\}$ , where  $l(b_1) \leq l(b'_1)$ ; and if  $n'(c) = 1$ , let  $S'(c) = \{c_1\}$ , otherwise let  $c_1$  be blank and  $l(c_1) = 0$ .

There are many possible arrangements to choose from, and the subcases require careful analysis.

Case C3.4.4.1,  $l(b_1) > m$ . If  $l(c_1) > m$ , set  $\varphi' = \alpha_{-a_m} \dots a_2 c a c_1 b$ ,  
If  $l(c_1) = m$ , set  $\varphi' = \alpha[c_1 a] c b$ . Otherwise set  
 $\varphi' = \alpha[ca]_b$ .

Case C3.4.4.2,  $l(b_1) \leq m$ ,  $l(c_1) \leq m$ . If  $l(c_1) = m$  or  $l(b'_1) < \infty$ ,  
set  $\varphi' = \alpha[c_1 a] c b$ . Otherwise if  $l(c_1) < l(b_1) - 2$ ,  
set  $\varphi' = \alpha[b_1 a] b c$ ; otherwise set  $\varphi' = \alpha[ca] b_1 b$ .

If  $l(b'_1) < \infty$ , success is imminent, so we may assume that  $l(b'_1) = \infty$ .  
Then  $\alpha[b_1 a] b c b'_1 \geq \alpha[ba] b'_1 c$ ; and  $\alpha[ca]_b \geq \alpha[c_1 a] c b \geq \alpha_{-a_m} \dots a_2 c a c_1 b$ ,  
unless  $l(c_1) = m$  when  $\alpha[ca]_b$  is inapplicable. If  $l(c_1) = m$ , it is  
clear that  $\alpha[c_1 a] c b \geq \alpha[b_1 a] b c b'_1$ ; otherwise we need to compare  
 $\alpha[b_1 a] b c b'_1$  with  $\alpha[ca]_b$ , and the best place for  $b_1$  in the latter  
string is  $\alpha[ca] b_1 b b'_1$ . The stretched chains in these two alternatives  
now fill respectively  $l(c_1)$  and  $l(b_1) - 2$  positions to the right of  $b'_1$ ,  
and it is best to minimize this quantity.

Case C3.4.4.3,  $l(b_1) \leq m$  and  $l(c_1) > m$ . If  $l(b_1) = m$ , set  
 $\varphi' = \alpha[b_1 a] b c b'_1 c_1$ . Otherwise if  $l(b'_1) = m$ , set  
 $\varphi' = \alpha[b'_1 a] b c b_1 c_1$ . Otherwise if  $l(b'_1) < m$ , set  
 $\varphi' = \alpha[b'_1 a] b_1 c$ . Otherwise let  $k = l(b_1)$ ; set  
 $\varphi' = \alpha_{-a_m} \dots a_{k+2} \langle b_k a_{k+1} \rangle \dots \langle b_1 a_2 \rangle \langle b a_1 \rangle \langle b'_1 c \rangle$ .

As in Case C3.4.2.2, this is a slight abuse of notation.

Case C3.4.5,  $S'(b) \cap S'(c) = \emptyset$ ,  $\max(n'(b), n'(c)) = 1$ . If  
 $n'(b) = 1$ , let  $S'(b) = \{b_1\}$ ; otherwise let  $b_1$   
be blank and set  $l(b_1) = 0$ . Define  $c_1$  similarly.

Case C3.4.5.1,  $l(b_1) \leq m$  and  $l(c_1) \leq m$ . Set  $\varphi' = \varphi b c$ .

Success is imminent.

Case C3.4.5.2,  $l(b_1) \leq m$  and  $l(c_1) > m$ . If  $l(b_1) = m$ , set  
 $\varphi' = \alpha[b_1 a] b c$ , otherwise set  $\varphi' = \alpha[ba]_c$ .

Case C3.4.5.3,  $l(b_1) > m$  and  $l(c_1) > m$ .

In this final case we must "look ahead" before deciding what to do.

For  $k \geq 1$  if  $b_k$  has degree 2, let  $b_{k+1}$  be the vertex adjacent to  $b_k$  which has not yet been given a name; continue until having found the sequence  $b - b_1 - \dots - b_k$  where  $\deg(b_k) \neq 2$ . Similarly, find the sequence  $c - c_1 - \dots - c_\ell$  where  $\deg(c_\ell) \neq 2$ .

(This process must terminate, since  $G$  is not a cycle.)

Case C3.4.5.3.1,  $b_k = c_\ell = a$  or  $\deg(b_k) = \deg(c_\ell) = 1$ . Set  $\varphi' = \varphi bc$ .

Success is imminent.

Case C3.4.5.3.2,  $\deg(b_k) = 1$  and  $\deg(c_\ell) > 2$ . Set  $\varphi' = \alpha_{-a_m} \dots a_2 bab_1 c$ .

Case C3.4.5.3.3,  $\deg(b_k) > 2$ ,  $\deg(c_\ell) > 2$ , and  $k \leq \ell$ .

In this case we must decide between four alternatives  $abcb_1c_1 \dots b_{k-1}c_{k-1}b_k$ ,  $acbc_1b_1 \dots c_{k-1}b_{k-1}c_k$ ,  $bab_1cb_2c_1 \dots b_kc_{k-1}$ , and  $cac_1bc_2b_1 \dots c_kb_{k-1}$ ; by acquiring a little more information about  $b_k$ ,  $c_\ell$ ,  $k$ , and  $\ell$  it will become clear which of these dominates:

Case C3.4.5.3.3.1,  $b_k = c_\ell$ . If  $k = \ell$ , set  $\varphi' = \varphi bcb_1c_1$ ; otherwise set  $\varphi' = \alpha_{-a_m} \dots a_2 ca_1c_1b$ .

Case C3.4.5.3.3.2,  $b_k \neq c_\ell$ . If  $k = \ell$ , set  $\varphi' = \varphi \langle bc \rangle \langle b_1c_1 \rangle$ ; otherwise set  $\varphi' = \alpha_{-a_m} \dots a_2 \langle ac \rangle \langle bc_1 \rangle$ .

Case C3.4.5.3.3.3,  $b_k \neq c_\ell$ ,  $b_k \neq c_\ell$ . Failure.

Note that the "lookahead time" required to find  $k$  and  $\ell$  in Case C3.4.5.3 is  $O(k+\ell)$ , not  $O(1)$ ; but Case C3.4.5.3 cannot occur again until  $b_1, \dots, b_{k-1}, c_1, \dots, c_{\ell-1}$  have all been included in the string  $\varphi$ . Thus the lookahead time can be distributed among the subsequent steps, and the subalgorithm runs in linear time.

We have now exhausted all possible cases, and the subalgorithm is complete.

## 6. Examples.

Here is how the subalgorithm would proceed to search for a layout for the graph of Figure 1, beginning with DC :

Case	$\varphi$
A3	DC
B2	DC(AE)
A3	DCAEBF
B1	DCAEBF(GJ)
	Failure.

On the other hand, if we begin with DA , the algorithm succeeds:

	DA
A2	DAC
A2	DACB
A2	DACBE
A4	DACBE F
C3.4.4.1(i)	DACBEGFHJ
A2	DACBEGFHJI
A2	DACBEGFHJIK
A4	DACBEGFHJIK L
C3.4.4.1(ii)	DACBEGFHJIKNLMO
A3	DACBEGFHJIKNLMO(PR)
B5	DACBEGFHJIKNLMO(PR)(QS)
B6	DACBEGFHJIKNLMOFQRS T
C4	DACBEGFHJIKNLMOFQRS T U
C2.3.1(ii)	DACBEGFHJIKNLMOFQRSWTUV(XY)
B2	DACBEGFHJIKNLMOFQRSWTUVYX&Z
A5	Success.

Here is how the algorithm would construct the same solution

"backwards", starting with Z& :

	Z&
A2	Z&X
A2	Z&XY
A2	Z&XYU
A3	Z&XYU(TV)
B5	Z&XYU(TV)(SW)
B6	Z&XYUVTWS R
C4	Z&XYUVTWS R O
C3.4.4.2(iii)	Z&XYUVTWSQRPOML
A2	Z&XYUVTWSQRPOMLN
A2	Z&XYUVTWSQRPOMLNK
A4	Z&XYUVTWSQRPOMLNK J
C3.4.4.1(i)	Z&XYUVTWSQRPOMLNKIJHF
A2	Z&XYUVTWSQRPOMLNKIJHFG
A2	Z&XYUVTWSQRPOMLNKIJHFGE
A3	Z&XYUVTWSQRPOMLNKIJHFGE(BC)
B2	Z&XYUVTWSQRPOMLNKIJHFGEBCAD
A5	Success.

If the algorithm had chosen the somewhat tempting alternative  
 Z&XYUVTWSNRMOLPK at step C3.4.4.2 in this example, failure would have  
 followed soon after.

Suppose Figure 1 were changed so that  $F - J$  became  $F - * - J$ .  
 Then the algorithm would invoke further cases:

C3.4.5.3.3.1(ii)	Z&XYUVTWSQRPOMLNK J
A2	Z&XYUVTWSQRPOMLNKIJH*
A2	Z&XYUVTWSQRPOMLNKIJH*G
A4	Z&XYUVTWSQRPOMLNKIJH*GF
C3.4.2.3	Z&XYUVTWSQRPOMLNKIJH*GF E
A5	Z&XYUVTWSQRPOMLNKIJH*GFDECBA
	Success.

## 7. Applications of the Subalgorithm.

The subalgorithm determines in  $O(n)$  steps whether or not  $G$  has a bandwidth-2 layout beginning with  $ab$ ; by trying all possible  $a$  and  $b$  we have an  $O(n^3)$  algorithm for deciding whether or not  $\text{Bandwidth}(G) \leq 2$ . This can be improved to an  $O(n^2)$  algorithm, by using the subalgorithm to decide whether or not  $G$  has a complete layout that extends  $xy_a$ , for some vertex  $a$  and some (nonexistent) dummy vertices  $x$  and  $y$ . However, we really want an  $O(n)$  algorithm, so it is necessary to be a little more careful.

We observed at the beginning of Section 2 that  $G$  may be assumed to contain a vertex  $v$  of degree  $\geq 3$ ; suppose  $v - a$ ,  $v - b$ , and  $v - c$ . Then any layout for  $G$  must contain one of the six substrings

$vab$ ,  $vba$ ,  $vac$ ,  $vca$ ,  $vbc$ ,  $vcb$ ,

or their left-right reflections, since two of  $\{a, b, c\}$  must appear on the same side of  $v$ . To test  $\text{Bandwidth}(G) \leq 2$  in linear time, it therefore suffices to have a linear-time algorithm that determines whether or not a complete layout exists containing a given substring of three vertices. (Recall that a "complete layout" always has bandwidth 2 according to the definition in Section 2.)

Let us first develop an algorithm which decides in  $O(n)$  steps whether or not there is a complete layout for a given connected graph  $G$ , containing a given substring  $abcd$  of length 4:

Step 1. Stop with failure if  $a - d$ .

Step 2. Let  $G_0$  be the graph obtained from  $G$  by deleting all edges among  $\{a, b, c, d\}$ . If there is a path in  $G_0$  from  $a$  or  $b$  to  $c$  or  $d$ , stop with failure. (This path cannot possibly be incorporated into a complete layout containing  $abcd$ , since it cannot get to the right of  $b$ .)

Step 3. Let the vertices of  $V \setminus \{a, b, c, d\}$  be partitioned into two subsets

$$V_1 = \{v \mid \text{a path exists in } G_0 \text{ from } v \text{ to } a \text{ or } b\},$$

$$V_2 = \{v \mid \text{a path exists in } G_0 \text{ from } v \text{ to } c \text{ or } d\}.$$

(By step 2,  $V_1$  and  $V_2$  are disjoint. Furthermore  $V = \{a, b, c, d\} \cup V_1 \cup V_2$ , since  $G$  was connected.) Let

$G_1$  be  $G_0$  restricted to  $V_1 \cup \{a, b\}$ , and let  $G_2$  be  $G_0$  restricted to  $V_2 \cup \{c, d\}$ . Use the subalgorithm to find a layout  $\phi_1$  for  $G_1$  beginning with  $ba$ , and also to find a layout  $\phi_2$  for  $G_2$  beginning with  $cd$ . If either attempt fails, stop with failure; otherwise stop with success, since  $\phi_1^R \phi_2$  is a complete layout for  $G$  as required.  $\square$

Now to solve the similar problem given a substring  $abc$  of length 3, we consider two cases:

- (i) There is at least one vertex  $d \neq a, c$  such that  $b - d$ . Then the complete layout must contain either  $abcd$  or  $dabc$ , and we use the previous algorithm to try both cases.
- (ii) There is no vertex  $d \neq a, c$  such that  $b - d$ . Then we can use an algorithm analogous to the one above: Let  $G_0$  be  $G$  minus all edges among  $\{a, b, c\}$  and stop if there is a path from  $a$  to  $c$  in  $G_0$ . Otherwise partition  $V \setminus \{a, b, c\}$  into disjoint sets  $V_1$  and  $V_2$ , where  $V_1$  contains the vertices reachable from  $a$  and  $V_2$  those reachable from  $c$ . Any complete layout containing the substring  $abc$  must be composed of a complete layout for  $G_1$  ending with  $ab$  and a complete layout for  $G_2$  beginning with  $bc$ .

It is also possible to construct a linear-time algorithm that decides whether or not a complete layout exists containing a given substring  $ab$  of length 2; details are left to the reader.

### 8. Tree Bandwidth is NP-complete.

In this section we shall prove that the general problem of determining the bandwidth of a graph is NP-complete; that is, any problem in the large class NP can be transformed into the problem of determining whether or not the bandwidth of some graph is less than some integer  $k$ , with at most a polynomial increase in the size of the problem specification. (See [25] and [2, Chapter 10] for surveys of NP-complete problems.) This particular result was first obtained by C. H. Papadimitriou [28]; we shall prove it in a sharper form, by severely restricting the form of  $G$ .

Theorem. The following problem is NP-complete: Given an integer  $k$ , and given a graph  $G$  which is a free tree with no vertices of degree  $> 3$ , is  $\text{Bandwidth}(G) \leq k$ ?

Proof. The problem of determining whether or not  $\text{Bandwidth}(G) \leq k$ , given  $k$  and an arbitrary graph  $G$ , is clearly in NP. We shall complete the proof by showing that the "3-partition problem," which is known to be NP-complete [16, p. 120], can be polynomially transformed into the restricted bandwidth problem stated in the theorem.

Given a sequence of  $3n$  integers  $\langle a_1, a_2, \dots, a_{3n} \rangle$ , where  $a_1 + a_2 + \dots + a_{3n} = nA$  and  $A/4 < a_i < A/2$  for each  $i$ , the 3-partition problem asks whether or not there is a way to partition the integers  $\{1, 2, \dots, 3n\}$  into disjoint triples  $T_1, \dots, T_n$  so that  $\sum \{a_j \mid j \in T_i\} = A$  for  $1 \leq i \leq n$ . In other words it is a special bin-packing problem, where we are to take  $3n$  objects of integer sizes  $a_1, a_2, \dots, a_{3n}$  and pack them into  $n$  boxes of size  $A$  whenever possible. The condition  $A/4 < a_i < A/2$  means that each box in any such packing must contain exactly three objects.

Given the specification of a 3-partition problem, our job is to construct an integer  $k$  and a free tree  $G$  whose vertices all have degree  $\leq 3$ , such that there is a 3-partition if and only if  $\text{Bandwidth}(G) \leq k$ . From the proof in [14] it suffices to do this with a tree whose size is at most a polynomial in  $n$  and  $A$ , since the 3-partition problem is NP-complete even when the magnitudes of all  $3n$  numbers are bounded above by a (suitably large) polynomial function of  $n$ . (See [15] for a discussion of this "strong NP-completeness" property.)

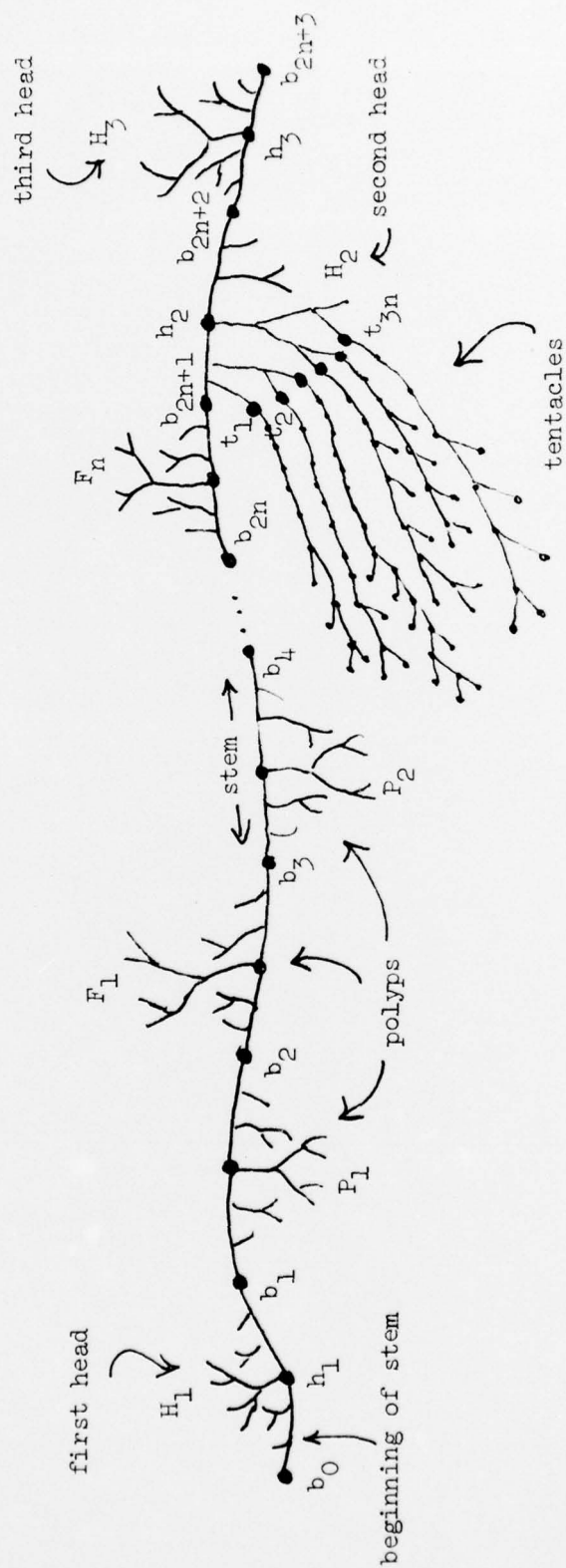


Figure 3. Siphonophore graph corresponding to 3-partition problem.

The free trees we shall construct bear more resemblance to pelagic hydrozoa of the order Siphonophora than to actual trees, so we shall find it convenient to use terms from marine biology rather than botany. Our construction involves parameters  $m_1, \dots, m_{3n}$ ,  $d$ , and  $k$  which we shall specify later after the properties we need for the proof have been explained.

The graphs of interest to use all have the general structure shown in Figure 3. There is a long stem, a path in which every  $d$ -th vertex has a special name; the respective names of these special stem vertices are

$$b_0 \ h_1 \ b_1 \ p_1 \ b_2 \ f_1 \ b_3 \ p_2 \ b_4 \ f_2 \ \dots \ p_n \ b_{2n} \ f_n \ b_{2n+1} \ h_2 \ b_{2n+2} \ h_3 \ b_{2n+3}$$

from left to right. It follows that the stem contains  $4dn + 6d + 1$  vertices in all. There are also  $3n$  long tentacles attached to special vertices  $t_1, \dots, t_{3n}$ ; the  $i$ -th tentacle consists of a long filament followed by  $2m_i$  nematocysts as shown in Figure 4. If we break off each tentacle just below the node  $t_i$ , and if we remove the boundary nodes  $b_0, b_1, \dots, b_{2n+3}$ , the remaining graph consists of  $2n+3$  connected pieces called polyps, named respectively

$$H_1 \ P_1 \ F_1 \ P_2 \ F_2 \ \dots \ P_n \ F_n \ H_2 \ H_3$$

from left to right. Note that the vertices  $t_1, \dots, t_{3n}$  all belong to the polyp called  $H_2$ , the animal's "second head".

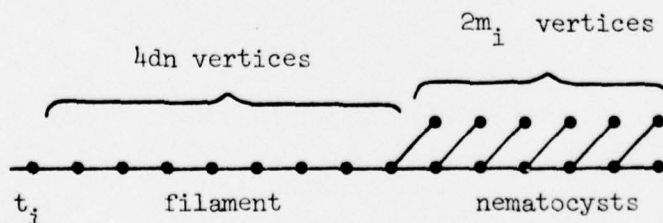


Figure 4. General form of the  $i$ -th tentacle.

We have noted that the special vertices  $b_0, h_1, b_1, \dots, b_{2n+3}$  are separated by distance  $d$ ; our construction will also have the property that every node of a polyp  $H_i$ ,  $P_i$ , or  $F_i$  is distance  $\leq d$  from its "central" node  $h_i$ ,  $p_i$ , or  $f_i$ .

Now we shall impose further constraints on the construction, so that it will not be easy to make layouts of bandwidth  $k$ . In the first place, we will require each of the heads  $H_i$  to contain exactly  $2dk-1$  vertices. This means that there are exactly  $2dk$  vertices  $\neq h_i$  at distance  $\leq d$  from  $h_i$  (since each head touches two boundary nodes  $b_j$ ), so it is necessary to lay these vertices out in such a way that the  $dk$  nearest locations on each side of  $h_i$  are occupied by precisely those elements at distance  $d$  or less in the graph. In particular, consider the layout of  $H_1$ , and assume without loss of generality that vertex  $b_1$  occurs to the right of  $h_1$ ; then all of the other polyps must appear to the right of  $H_1$  in the layout, since there is no way for any of their vertices to get to the left of  $h_1$  without making the bandwidth  $> k$ . A similar argument applies to the third head  $H_3$ , which therefore must appear (together somehow with  $b_{2n+3}$ ) at the extreme right of the layout. All of the other polyps, and all of the tentacles, must appear between  $H_1$  and  $H_3$ .

We shall arrange things so that the total number of vertices in the graph is exactly  $(2n+3)(2dk)+1$ . This means that the situation will be very "tight": There are  $(2n+1)(2dk)-1$  vertices which must appear in the layout between  $b_1$  and  $b_{2n+2}$ , but vertices  $b_1$  and  $b_{2n+2}$  are at distance  $(2n+1)(2d)$  from each other in the graph, so we must conclude that the stem between  $b_1$  and  $b_{2n+2}$  is stretched tightly. In other words, two adjacent nodes in this portion of the stem must be placed  $k$  positions apart. (It does not follow that the stem from  $b_0$  to  $h_1$  or from  $h_3$  to  $b_{2n+3}$  is stretched;  $b_0$  might even appear to the right of  $h_1$ . But all we are using  $H_1$  and  $H_3$  for is to confine the other nodes and therefore to assign a rigid structure to the interior parts of the layout.)

Since the stem is stretched tightly, and since the polyps contain no nodes at distance  $> d$  from their central node, the layout must now appear as a sequence of regions which we may represent as follows:

$$H'_1 b_1 P'_1 b_2 F'_1 b_3 P'_2 b_4 F'_2 \dots P'_n b_{2n} F'_n b_{2n+1} H'_2 b_{2n+2} H'_3 .$$

Here  $H'_1$  is a layout of  $H_1 \cup \{b_0\}$ ,  $H'_2$  is a layout of  $H_2$ ,  $H'_3$  is a layout of  $H_3 \cup \{b_{2n+3}\}$  and  $(P'_i, F'_i)$  are respectively layouts of  $(P_i, F_i)$  plus portions of the tentacles which just manage to fit. Each of the regions  $P'_i, F'_i$  includes exactly  $2dk-1$  vertices of the layout. The reader should stop at this point to review the construction before going on.

If we choose the sizes of  $P_i, F_i$  carefully it will be difficult to place the tentacles. Let us say that

$F_i$  contains exactly  $2dk-1-6di$  vertices,

$P_i$  contains exactly  $2dk-1-c-18di+12d$  vertices,

so that

$F'_i$  contains exactly  $6di$  tentacle vertices,

and  $P'_i$  contains exactly  $c+18di-12d$  tentacle vertices,

where  $c$  is a constant to be determined later. Note that the tentacles are all connected to  $H_2$ , so they have to emanate from near the right end of the layout, passing through  $F'_i$  before coming to  $P'_i$ . If  $P'_1, \dots, P'_i$  together contain portions of at least  $r_i$  different tentacles then  $F'_i$  must contain at least  $2dr_i$  vertices of these tentacles, since a path cannot cross  $F'_i$  without using up at least  $2d$  positions; hence  $2dr_i \leq 6di$ , i.e.,

$$r_i \leq 3i .$$

Furthermore if  $r_i = 3i$  each tentacle must use exactly  $2d$  positions of  $F'_i$ , so there can be no nematocysts in  $F'_i$  in this case.

By choosing the values of  $c, m_1, \dots, m_{3n}$  we will be able to guarantee that exactly  $3i$  tentacles come through  $F'_i$ . Consider first  $P'_1$ , which must contain  $c+6d$  tentacle vertices; these must come from at most three different tentacles because of the constraint on  $r_1$ . If we choose each  $m_i$  as a function of the given numbers  $a_i$  so that the number of nodes in two tentacles is always less than  $c+6d$ , then  $P'_1$  must contain vertices from exactly three different tentacles, and it must include all of their

nematocysts too because of the constraint on  $F'_1$ . Furthermore we will be able to argue in the same way that  $P'_2$  must now include all the nematocysts of three other tentacles because of the constraint on  $F'_2$ , and so on.

In order to make this argument go through properly we will want to define things so that the three tentacles whose nematocysts appear in  $P'_i$  have their filaments "pulled completely through" the succeeding regions, with exactly  $2d$  vertices of their filaments appearing in each of  $F'_i, P'_{i+1}, \dots, P'_n, F'_n$ . It turns out that we can do this by making each  $m_i$  a multiple of  $6dn$ , and requiring that  $a_i + a_j + a_\ell = A$  if and only if  $2(m_i + m_j + m_\ell) = c$ . Let us set

$$m_i = 6dna_i, \quad c = 12dnA;$$

we shall prove that a layout of bandwidth  $k$  implies the existence of a 3-partition:

Lemma. For  $1 \leq i \leq n$ , region  $P'_i$  contains all of the nematocysts from exactly three tentacles, namely the tentacles connected to  $t_j$  where  $j$  is in some triple  $T_i$ , and  $\sum \{a_j \mid j \in T_i\} = A$ . Furthermore  $P'_i$  also contains as much as possible of the filaments from these tentacles, i.e., each tentacle in  $T_i$  has only  $2d$  vertices in each of  $F'_i, P'_{i+1}, \dots, P'_n, F'_n$ .

Proof. By induction on  $i$ , we know that  $F'_i$  and  $P'_i$  each contain  $3(i-1)(2d)$  filament nodes from tentacles whose nematocysts appear in  $P'_1 \dots P'_{i-1}$ . That leaves  $6d$  empty positions in  $F'_i$  and  $12dnA + 12di - 6d$  in  $P'_i$ . Now  $P'_i$  must contain vertices from at least three tentacles, since two tentacles have at most  $8dn + 2(m_j + m_\ell) = 8dn + 12dn(a_j + a_\ell) \leq 8dn + 12dn(A-1) = 12dnA - 4dn$  vertices altogether.

Hence  $P'_i$  has vertices from exactly three tentacles, defined by some triple  $T_i \subseteq \{1, 2, \dots, 3n\}$ , and it includes all of their nematocysts because  $F'_i$  has room for only  $6d$  more vertices from all three tentacles. Let  $\sum \{a_j \mid j \in T_i\} = \alpha$ ; then the  $12dnA + 12di - 6d$  available positions in  $P'_i$  are taken up by  $12dn\alpha$  nematocysts and somewhere between  $0$  and  $3(4dn - (2n - 2i + 1)(2d)) = 12di - 6d$  filament nodes. It follows that  $\alpha = A$

and exactly  $12di \ 6d$  filament nodes are present.  $\square$

The lemma proves that a bandwidth- $k$  layout for a graph of this kind necessarily leads to a valid 3-partition. To complete the proof of the theorem, we must define the graphs so that existence of a 3-partition is sufficient to imply the existence of a layout with bandwidth  $k$ . This means in particular that we will have to choose  $d$  and  $k$  appropriately. Furthermore the graphs must be constructible by an algorithm whose running time is bounded by a polynomial in  $n$  and  $A$ .

In the first place we want to choose  $k$  large enough that  $P_n$  contains at least  $2d-1$  vertices, hence we require

$$k \geq 6nA + 9n - 5.$$

For convenience we let  $k$  be the smallest power of 2 satisfying this condition, and we write

$$k = 2^l.$$

Finally we choose

$$d = lk.$$

From these parameters  $k$  and  $d$  we can construct  $G$  by explaining how to construct each polyp. The head polyps  $H_i$  are formed by the bandwidth- $2^l$  layout indicated in Figure 5 for  $l = 3$  (although  $l$  will never be this small). A periodic pattern begins to repeat after the  $l$ -th stem node to the right of  $h_i$ : the  $j$ -th node preceding a stem node branches to the  $(2j)$ -th and  $(2j+1)$ -st nodes preceding the next stem node, for  $0 \leq j < 2^{l-1}$ . Before this pattern is established, we have  $(1, 2, 4, \dots, 2^{l-2})$  as the respective limits on  $j$ . An additional "thread branch" goes out of  $h_i$  to fill up the remaining  $(2^l - 1) + (2^l - 2) + \dots + (2^l - 2^{l-1}) = lk - 2^{l+1} = d - k + 1$  holes near the center. To the left of  $h_i$  we use essentially the same idea in mirror image; thus it is clear that no vertex is at distance greater than  $d$  from the center node. The special nodes  $t_1, \dots, t_{3n}$  in  $H_2$  are taken to be the leftmost  $3n$  nodes in its layout.

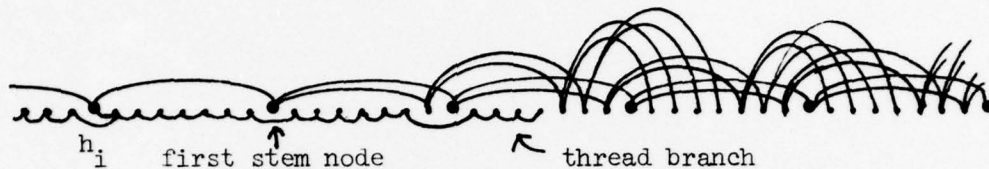


Figure 5. Layout of a head polyp  $H_i$  in the immediate vicinity of its center node  $h_i$ .

A similar procedure is used to construct the other polyyps  $P_i$  and  $F_i$ . In each case we wish to remove  $2dx$  nodes from a full head polyp, for some integer  $x$ , and we do this by removing  $x$  nodes between each pair of adjacent stem nodes. The  $x$  nodes immediately to the right of each stem node in Figure 5 are simply deleted from the graph, together with all edges touching them, and the "thread branch" is reconnected for the remaining nodes; again the mirror image of this pattern is used to the left of the center vertex, and we clearly have a tree. It is easy to see that the resulting polyp has a layout of length  $2dk-1$  in which the  $x$  positions just to the left of each stem node are empty. (Simply shift all non-stem vertices which lie to the right of the center vertex exactly  $x$  places to the left.) These  $x$  slots form  $x$  parallel "channels" through which filaments can pass.

Now it is not difficult to see how to embed the tentacles into these polyp layouts whenever a 3-partition is given. For example, we can place filaments for the three tentacles specified by  $T_1$  into the rightmost three channels of  $F_1, P_2, F_2, \dots, P_n, F_n$ . Now it is easy to make the remaining nematocyst and filament nodes fit into the remaining spaces in  $P_1$  without

exceeding bandwidth  $k$  ; further details are left to the reader. It is possible to link up any channel in  $F_n$  with any  $t_i$  , since  $k' \geq 6n$  .  $\square$

#### 9. Directed Bandwidth.

Analogous problems can be studied when  $G$  is an acyclic directed graph, where we require its layout to be a topological sorting of the vertices; in other words, we stipulate that  $f(u) < f(v)$  whenever  $u \rightarrow v$  in the graph, and we ask for the minimum bandwidth subject to this constraint.

The algorithm in Sections 2 through 7 above can readily be modified to test for "directed bandwidth 2." In fact, the situation becomes so much simpler that it is tempting to try for directed bandwidth 3 in polynomial time.

The NP-completeness construction in Section 8 can be modified in a straightforward way to obtain an analogous result.

Theorem. The following problem is NP-complete: Given an integer  $k$ , and given a directed graph which is an oriented tree having no vertices of in-degree  $> 2$ , is its directed bandwidth  $\leq k$ ?

(Each vertex of an oriented tree has out-degree  $\leq 1$ , and there are no cycles.)

The analogous problem of minimizing  $\sum (f(v) - f(u))$  over all topological sortings of a general acyclic directed graph has recently been proved NP-complete by E. L. Lawler [26]; on the other hand Adolphson and Hu [1] have resolved this problem in polynomial time when the directed graph is an oriented tree, even when the arcs have been assigned arbitrary weights. The above theorem indicates that the bandwidth problem is somewhat harder than this optimal ordering problem, in the directed as well as the undirected case.

#### 10. Some Open Problems.

The following related questions are still waiting for an answer:

- (a) Is the problem " $\text{Bandwidth}(G) \leq 3$ " NP-complete, given an arbitrary graph (or perhaps a tree)  $G$ ?
- (b) Is there a polynomial time algorithm to enumerate the number of distinct bandwidth-2 layouts of a given graph  $G$ ?
- (c) For which exponents  $m$  is the problem "Some layout of  $G$  satisfies  $\sum \{|f(u) - f(v)|^m : u - v \text{ in } G\} \leq k$ " NP-complete, when  $G$  is a free tree?
- (d) What is the expected minimum bandwidth, for random graphs on  $n$  vertices and  $m$  edges, as  $n$  and  $m \rightarrow \infty$ ?

Question (b) is of potential interest because there seems to be a vague connection between efficient algorithms for enumeration and efficient algorithms for testing existence. For example, there is a determinant formula for evaluating the number of spanning trees of a graph, and there are efficient algorithms for testing connectedness. The problems of enumerating the number of hamiltonian paths of a graph, or the number of ways to satisfy a given set of clauses, etc., do not seem to be in NP; there most likely are polynomial-time reducibilities between such problems, but such transformations remain to be investigated. In the case of bandwidth-2 layouts for a graph, there is a linear time algorithm for existence, yet no apparently "nice" characterization. So this is a candidate problem in which enumeration might be definitely more difficult than existence.

Question (c) is suggested by the observation that the stated problem is solvable in polynomial time for  $m = 1$  [31], but as  $m$  increases the best layouts are eventually those with minimum bandwidth.

All four problems can be considered also for the case of directed bandwidth.

Another interesting question is to discover how far from optimum the various heuristic methods for bandwidth reduction can be; see the references below for several approaches that have been proposed.

# References

- [1] D. Adolphson and T. C. Hu, "Optimal linear ordering," SIAM J. Appl. Math. 25 (1973), 403-423.
- [2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, (Reading, Mass.: Addison-Wesley, 1974), x+470 pp.
- [3] F. A. Akyuz and S. Tuku, "An automatic node-relabelling scheme for bandwidth minimization of stiffness matrices," Amer. Inst. of Aero. and Astro. Journal 6 (1968), 728-730.
- [4] G. G. Alway and D. W. Martin, "An algorithm for reducing the bandwidth of a matrix of symmetrical configuration," Comp. J. 8 (1965), 264-272.
- [5] I. Arany, L. Szoda, and W. F. Smyth, "An improved method for reducing the bandwidth of sparse symmetric matrices," Proc. IFIP Conference 1971 (North-Holland Publishing Company, 1972), 1246-1250.
- [6] J. Bolstad, G. Leif, A. Lindeman, and H. Kaper, "An empirical investigation of reordering and data management for finite element systems of equations," Argonne report ANL8056, September 1973, 50 pp.
- [7] K. Y. Chen, "Minimizing the bandwidth of sparse symmetric matrices," Computing 11 (1973), 27-30, 103-110.
- [8] V. Chvátal, "A remark on a problem of Harary," Czechoslovak Math. J. 20 (1970), 109-111.
- [9] Jarmila Chvátalová, "Optimal labelling of a product of two paths," Discrete Math. 11 (1975), 249-253.
- [10] J. Chvátalová, A. K. Dewdney, N. E. Gibbs, and R. R. Korfhage, "The bandwidth problem for graphs, a collection of recent results," Research report no. 24, Dept. of Computer Science, Univ. of Western Ontario (London, Ontario, Canada, 1975).
- [11] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," Proc. ACM National Conference 24 (1969), 157-172.
- [12] Richard A. DeMillo, Stanley C. Eisenstat, and Richard J. Lipton, "Preserving average proximity in arrays," Dept. of Computer Science, Yale University, TR-CS-76-4 (March 1976), 10 pp.
- [13] D. R. Fulkerson and O. A. Gross, "Incidence matrices and interval graphs," Pacific J. Math. 15 (1965), 835-855. [Generalized bandwidth-1 problem for hypergraphs.]

- [14] M. R. Garey and D. S. Johnson, "Complexity results for multiprocessor scheduling under resource constraints," SIAM J. Computing 4 (1975), 397-411.
- [15] M. R. Garey and D. S. Johnson, "'Strong' NP-completeness results: Motivation, examples and implications," submitted for publication.
- [16] M. R. Garey, D. S. Johnson, and Ravi Sethi, "The complexity of flowshop and jobshop scheduling," Mathematics of Operations Research 1 (1976), 117-129.
- [17] M. R. Garey, D. S. Johnson, and L. Stockmeyer, "Some simplified NP-complete graph problems," Theoretical Computer Science 1 (1976), 237-267.
- [18] J. Alan George, "Computer implementation of the finite element method," Ph.D. Thesis, Computer Science Department, Stanford University, March 1971, 220 pp.
- [19] Norman E. Gibbs and William G. Poole Jr., "Tridiagonalization by permutations," Comm. ACM 20 (1974), 20-24.
- [20] R. L. Graham, "On primitive graphs and optimal vertex assignments," Ann. N. Y. Acad. Sci. 175 (1970), 170-186.
- [21] H. R. Grooms, "Algorithm for matrix bandwidth reduction," J. of the Structural Div., Proc. Amer. Soc. Civil Eng. 98 (1972), 203-214.
- [22] L. H. Harper, "Optimal assignments of numbers to vertices," J. Soc. Indust. Appl. Math. 12 (1964), 131-135.
- [23] L. H. Harper, "Optimal numberings and isoperimetric problems on graphs," J. Comb. Theory 1 (1966), 385-393.
- [24] L. H. Harper, "A necessary condition on minimal cube numberings," J. Appl. Prob. 4 (1967), 397-401.
- [25] R. M. Karp, "Reducibility among combinatorial problems," in Complexity of Computer Computations, ed. by R. E. Miller and J. W. Thatcher (New York: Plenum, 1972), 85-104.
- [26] E. L. Lawler, "Sequencing jobs to minimize total weighted completion time subject to precedence constraints," submitted for publication.
- [27] R. J. Lipton, S. C. Eisenstat, and R. A. DeMillo, "Space and time hierarchies for classes of control structures and data structures," J. ACM 23 (1976), 720-732. [Generalized bandwidth problem of embedding one graph in another.]

- [28] Christos H. Papadimitriou, "The NP-completeness of the bandwidth minimization problem," Computing 16 (1976), 263-270.
- [29] R. Rosen, "Matrix bandwidth minimization," Proc. ACM National Conference 23 (1968), 585-595.
- [30] A. L. Rosenberg, "Preserving proximity in arrays," SIAM J. Computing 4 (1975), 443-460.
- [31] Yossi Shiloach, "Optimal placement problems," Ph.D. Thesis, Feinberg Graduate School, Weizmann Institute (Rehovot, Israel, 1975).
- [32] Paul Ting Renn Wang, "Bandwidth minimization, reducibility decomposition, and triangularization of sparse matrices," Computer and Information Science Research Center, Ohio State University, report OSU-CISRC-TR-73-5 (September 1973), 162 pp.